Objective

- To learn more advanced features of the CUDA APIs for data transfer and kernel launch
  - Task parallelism for overlapping data transfer with kernel computation
  - CUDA streams
So far, the way we use `cudaMemCpy` serializes data transfer and GPU computation.

- Trans. A
- Trans. B
- Vector Add
- Tranfer C

Only uses one direction, GPU idle

PCIe Idle

Only uses one direction, GPU idle
Device Overlap

• Some CUDA devices support device overlap
  – Simultaneously execute a kernel while performing a copy between device and host memory

```c
int dev_count;
cudaDeviceProp prop;

cudaGetDeviceCount(&dev_count);
for(int i=0; i < dev_count; i++){
    cudaGetDeviceProperties(&prop, i);
    if (prop.deviceOverlap) ... 
```
Overlapped (Pipelined) Timing

- Divide large vectors into segments
- Overlap transfer and compute of adjacent segments

```
Trans A.1          Trans B.1          Comp C.1 = A.1 + B.1          Trans C.1
Trans A.2          Trans B.2          Comp C.2 = A.2 + B.2          Trans C.2
Trans A.3          Trans B.3          Comp C.3 = A.3 + B.3          Trans C.3
Trans A.4          Trans B.4
```
CUDA supports parallel execution of kernels and `cudaMemCpy` with “Streams”

- Each **stream** is a **queue** of operations (kernel launches and `cudaMemCpys`)
- Operations (tasks) in different streams can go in parallel
  - “Task parallelism”
Streams

- Device requests made from the host code are put into a queue
  - Queue is read and processed *asynchronously* by the driver and device
  - Driver ensures that commands in the queue are processed in sequence. Memory copies end before kernel launch, etc.

Data Transfer and CUDA Streams
To allow concurrent copying and kernel execution, you need to use multiple queues, called “streams”.

CUDA “events” allow the host thread to query and synchronize with the individual queues.
Data Transfer and CUDA Streams

**Conceptual View of Streams**

- **PCIe UP**
- **PCIe Down**
- **Copy Engine**
- **Kernel Engine**

- **MemCpy A.1**
- **MemCpy B.1**
- **Kernel 1**
- **MemCpy C.1**

- **MemCpy A.2**
- **MemCpy B.2**
- **Kernel 2**
- **MemCpy C.2**

**Stream 0**

**Stream 1**

Operations (Kernels, MemCpys)
A Simple Multi-Stream Host Code

cudaStream_t stream0, stream1;
cudaStreamCreate( &stream0 );
cudaStreamCreate( &stream1 );
float *d_A0, *d_B0, *d_C0; // device memory for stream 0
float *d_A1, *d_B1, *d_C1; // device memory for stream 1

// cudaMalloc for d_A0, d_B0, d_C0, d_A1, d_B1, d_C1 go here

for (int i=0; i<n; i+=SegSize*2) {
    cudaMemcpyAsync(d_A0, h_A+i, SegSize*sizeof(float),..., stream0);
    cudaMemcpyAsync(d_B0, h_B+i, SegSize*sizeof(float),..., stream0);
    vecAdd<<<SegSize/256, 256, 0, stream0>>;
    cudaMemcpyAsync(d_C0, h_C+i, SegSize*sizeof(float),..., stream0);
    ...
}
A Simple Multi-Stream Host Code

for (int i=0; i<n; i+=SegSize*2) {
    // stream 0
    cudaMemcpyAsync(d_A0, h_A+i, SegSize*sizeof(float), ..., stream0);
    cudaMemcpyAsync(d_B0, h_B+i, SegSize*sizeof(float), ..., stream0);
    vecAdd<<<SegSize/256, 256, 0, stream0>>>(d_A0, d_B0, ...);
    cudaMemcpyAsync(d_C0, h_C+i, SegSize*sizeof(float), ..., stream0);
    // stream 1
    cudaMemcpyAsync(d_A1, h_A+i+SegSize,
                    SegSize*sizeof(float), ..., stream1);
    cudaMemcpyAsync(d_B1, h_B+i+SegSize,
                    SegSize*sizeof(float), ..., stream1);
    vecAdd<<<SegSize/256, 256, 0, stream1>>>(d_A1, d_B1, ...);
    cudaMemcpyAsync(d_C1, h_C+i+SegSize,
                    SegSize*sizeof(float), ..., stream1);
}
A View Closer to Reality

Copy Engine

MemCpy A.1
MemCpy B.1
MemCpy C.1
MemCpy A.2
MemCpy B.2
MemCpy C.2

Kernel Engine

Kernel 1
Kernel 2

Dependencies

Stream 0

Operations (Kernels, MemCpys)

Stream 1
Not quite the overlap we want ...

- C.1 blocks A.2 and B.2 in the copy engine queue
A Better Multi-Stream Host Code

for (int i=0; i<n; i+=SegSize*2) {
    // enqueue A0, B0 --> A1, B1
    cudaMemCpyAsync(d_A0, h_A+i; SegSize*sizeof(float),..., stream0);
    cudaMemCpyAsync(d_B0, h_B+i; SegSize*sizeof(float),..., stream0);
    cudaMemCpyAsync(d_A1, h_A+i+SegSize,
                    SegSize*sizeof(float),..., stream1);
    cudaMemCpyAsync(d_B1, h_B+i+SegSize,
                    SegSize*sizeof(float),..., stream1);

    // enqueue kernel 0, kernel 1
    vecAdd<<<SegSize/256, 256, 0, stream0>>>(d_A0, d_B0, ...);
    vecAdd<<<SegSize/256, 256, 0, stream1>>>(d_A1, d_B1, ...);

    // enqueue C0 --> C1
    cudaMemCpyAsync(d_C0, h_C+i, SegSize*sizeof(float),..., stream0);
    cudaMemCpyAsync(d_C1, h_C+i+SegSize,
                    SegSize*sizeof(float),..., stream1);
}
A View Closer to Reality

Operations (Kernels, MemCpis)

Stream 0

MemCpy A.1
MemCpy B.1
MemCpy A.2
MemCpy B.2
MemCpy C.1
MemCpy C.2

Copy Engine

Kernel Engine

Kernel 1
Kernel 2

PCIe UP
PCIe Down

Copyright © 2013 by Yong Cao, Referencing UIUC ECE408/498AL Course Notes
Overlapped (Pieplined) Timing

- Divide large vectors into segments
- Overlap transfer and compute of adjacent segments

Trans A.1  |  Trans B.1  |  Comp C.1 = A.1 + B.1

Trans A.2  |  Trans B.2  |  Comp C.2 = A.2 + B.2

Trans A.3  |  Trans B.3  |  Comp C.3 = A.3 + B.3

Trans A.4  |  Trans B.4
Hyper Queue

- Provide **multiple real queues** for each engine
- Allow much more concurrency by allowing some **streams to make progress for an engine** while others are blocked
Fermi (and older) Concurrency

Fermi allows 16-way concurrency

- Up to 16 grids can run at once
- But CUDA streams multiplex into a single queue
- Overlap only at stream edges
Kepler Improved Concurrency

Kepler allows 32-way concurrency

One work queue per stream
Concurrency at full stream level
No inter-stream dependencies